Predicate Abstraction for Object-Oriented Programs

SeungJoon Park[1,3] Willem Visser[1,3] Phil Oh[2,3]

1. Research Institute for Advanced Computer Science (RIACS)
2. Caelum Research Corporation
3. Automated Software Engineering Group, NASA Ames Research Center
{spark,wvisser,oh}@ptolemy.arc.nasa.gov

Abstract. This paper presents an automated abstraction technique for abstracting software programs. The technique extends and applies predicate abstraction to general-purpose object-oriented programs. Each of state-transition statement in a concrete program is abstracted with respect to the user-guided predicates. The abstraction algorithm consists of the following steps: (1) Compute pre-images of abstraction predicates with respect to a given statement, (2) Map the pre-images into abstract domain, (3) Generate an abstract statement which sets the values of abstract variables using the conditions of the abstracted pre-images. An automated decision procedure which checks the validity of logical expressions is used to compute sound abstraction automatically. This paper also proposes a solution on the use of predicate abstraction in object-oriented paradigm. The techniques have been implemented in an automated prototype tool for Java which generates abstract programs at the source level.

1 Introduction

Model checking software systems often suffers from state space explosion and this problem is even more acute for checking actual source code. Experienced verification engineers may manipulate the code to obtain an abstracted system that can be model checked. However, such manual abstraction is very difficult and time consuming: it may require a lot of manual reasonings, which delays the verification process significantly, and the human reasonings can be error-prone leading to false positives.

This paper presents an automated abstraction technique for abstracting software programs. The technique extends and applies predicate abstraction to general-purpose object-oriented programs. Predicate abstraction is a mapping of a concrete state transition system to an abstract state transition system, whose state corresponds to the truth values of a set of predicates in concrete state. The abstraction technique provides a way

of combining theorem proving and model checking for the verification of unbounded systems.

Predicate abstraction was originally proposed to be used for constructing a reachable abstract state space on-the-fly [4, 3]. Unfortunately, software systems usually have very large state spaces even in the abstract domain, which is very expensive to search exhaustively using decision procedures. Moreover, computing the abstract state space on-the-fly makes it difficult to use state-of-the-art model checking techniques such as symmetry reduction, partial order reduction, or hash compaction.

Our approach to the verification is to generate an abstract program based on predicate abstraction then apply model checking. We abstract each of state-transition statement in a concrete program instead of computing the abstract state space of the program. Since each concrete statement is abstracted to an abstract statement, the abstract program would have a very similar class structures and control flows to those of the concrete program. This clear correspondence between abstract and concrete statements in the program helps the user to trace an error in concrete domain when model checking finds an error trace in the abstract program. In addition, the computation cost of the abstraction is proportional to the size of program rather than the size of the reachable state space.

We have been developing an automated prototype tool for Java which generates an abstract program at the source level. We allow the user to specify abstraction by removing concrete variables in the program and/or adding new abstract variables which are defined in terms of concrete variables. The predicate abstraction is used for mapping one or more variables in large or unbounded ranges to an abstract boolean variable. Once the abstraction specified, our tool automatically computes an abstract program with the new abstract variables and unremoved variables in the concrete program (current prototype tool supports adding boolean types only).

For a given Java program, the user provides predicate abstractions to have the tool construct an abstract Java program. Then, the user may apply a model checker to the abstract program to check corresponding properties in abstract domain. Our abstraction algorithm consists of the following steps: (1) Compute pre-images of abstraction predicates with respect to a given statement, (2) Map the pre-images into abstract domain, (3) Generate an abstract statement which sets the values of abstract variables using the conditions of the abstracted pre-images. To compute sound abstraction automatically, we use an automated decision procedure which checks the validity of logical expressions [1].

Our current prototype tool supports only a subset of Java syntax. It does not allow predicate abstraction depending on array variables or local variables. The tool does not support abstraction on the variables which are used as parameters for method calls. The tool requires the user provide the same predicate as in a Boolean condition in the program if he/she wants to abstract any part of the condition, though we expect this restriction will be lifted soon.

Related work

We believe that our work is the first application of the predicate abstraction to a real programming language. However, there has been similar work by others in different frames. Colon and Uribe [2] use decision procedures to generate finite-state abstractions of transitions of a system with respect to a set of predicates. Saïdi and Shankar [7] did similar work using PVS. Both of the work require use of only global variables to describe a system in simple languages similar to guarded commands.

We aim to abstract object-oriented programs and propose solutions for dealing with object-orientedness. In addition, we exploits pre-images of the given predicates for the computation of abstract statements. This enables us to obtain accurate abstract statement without using the two-folded state representation (in terms of the current and the previous states) as in [7].

The Bandera tool from Kansas University [5] uses techniques to abstract the data domain of a variable in Java program using PVS decision procedure. However, the technique is limited for reduction of the data range of a variable to a smaller range. The novelty of our approach lies in the fact that we can abstract a system with respect to predicates over multiple variables and even over more than one class.

Namjoshi and Kurshan [6] have focused on a refinement technique for finding predicate abstraction criteria automatically. They used simple syntactic rewriting techniques to compute abstract programs.

2 Abstracting Programs

The transition-based abstraction limits the scope of abstraction to each statement in the program, which could result in a less accurate abstraction than that based on state space abstraction. However, we obtain an as accurate abstract statement as possible by computing pre-images of the abstraction predicates with respect to a concrete statement and use them as guard conditions of the abstract state transition.

This section first explains the detailed abstraction algorithm on assignment statements, which are typical statements that change the state of the program. Then we extend the predicate abstraction to object-oriented programs.

2.1 Abstracting assignment statements

Once the user annotates abstraction by removing some of the concrete variables in the program and/or adding new abstract variables, the abstract program will have new abstract variables and unremoved variables in the concrete program. Suppose a program has an integer variable x and the user wants to abstract the program by removing the variable and instead adding a new boolean variable defined by a predicate $B \equiv x > 0$. In the abstract program, there will be a new boolean variable b whose truth value corresponds to the predicate B. Let us first consider an assignment statement $x = 2$.

A naive way to compute the abstract statement is to check the post-condition of the assignment. The post-condition $x = 2$ of the assignment implies the predicate B so the corresponding abstract statement would be $b = true$. What about another assignment $x = x + 1$? For this assignment, checking post-condition is not good enough to find the abstract assignment because the condition does not imply either B or $\neg B$, which would lead to an abstract statement setting the value of b to be nondeterministic.

However, we can use pre-images of the abstract predicate to obtain a more accurate abstract statement. A pre-image of a predicate with respect to a transition is a set of states which will lead to a state that satisfies the predicate by the transition. The pre-image of a predicate by an assignment can be easily computed by substituting the lefthand-side variable of the assignment by the righthand-side expression into the predicate. For example, the pre-image of B with respect to the transition $x = x+1$ is $x+1 > 0$ and that of $\neg B$ is $x+1 \leq 0$. Once we have pre-images, we can attempt to generate an abstract statement of the assignment,

$$\begin{array}{l}\text{if } (x + 1 > 0) \text{ then } b = \text{true;} \\ \text{else if } (x + 1 \leq 0) \text{ then } b = \text{false;}.\end{array} \qquad (1)$$

Unfortunately, the above abstract statement is not meaningful if the variable x needs to be abstracted out from the program. In such case, the pre-images in the concrete domain need to be mapped into corresponding conditions in the abstract domain. Because we aim to compute sound

over-approximation of the system, the pre-image conditions should be replaced by stronger conditions (under-approximation) in terms of abstract variables. An accurate mapping of concrete expression to abstract domain can be computed using the technique proposed in [3]. Intuitively, the condition $x + 1 > 0$ can be replaced by b because $B \Rightarrow x + 1 > 0$ is valid. However, there is no such stronger condition in terms of b that implies the pre-image, $x + 1 \leq 0$. Therefore, we set b to be a nondeterministic boolean value (we use a notation, randomBool[1]) for the latter case:

$$\begin{array}{l} \text{if } b \text{ then } b = \text{true;} \\ \text{else } b = \text{randomBool;}. \end{array} \qquad (2)$$

Figure 1 shows the abstraction algorithm for converting an assignment statement *lvar* = *rexp*. The abstract statement to be generated consists of two parts: one for the assignment to *lvar* itself and the other for the assignments to new abstract variables that depend on *lvar*. If the variable *lvar* in the assignment has been removed, then the first part is not necessary. Otherwise, we generate an assignment to *lvar* by the same *rexp* if the expression does not contain any removed variables, or by the truth value that still can be decided by a validity checker. Note that if *lvar* ranges over a large data domain (such as unbounded integer), assigning random value is not practical for model checking. In this case, our tool guides the user to remove the variable *lvar* and provide abstraction predicates which keep necessary information about the variable.

For each of abstraction predicate depending on *lvar*, a truth value is assigned to the Boolean variable. We first compute the pre-images of the predicate and the negation of the predicate with respect to the original assignment. Then we compute their under-approximation $U()$ in the abstract domain, and use them as guard conditions of the abstract assignment.

2.2 Conditions

Besides the state changing assignment statements, condition expressions in the program are also affected by the abstraction if they contain some variables to be removed. As our current tool does, a simple solution is to require the user specify predicate abstractions for such conditions in the program. Then the conditions in the program are replaced by the matching boolean variables in the abstract program.

[1] This will be recognized by a model checker as a nondeterministic choice when it explores the state space of the abstract program.

```
if lvar is not removed
  if rexp does not contain any removed variables
    generate "lvar = rexp;"
  else if lvar is boolean and rexp is valid
    generate "lvar = true;"
  else if lvar is boolean and ¬rexp is valid
    generate "lvar = false;"
  else
    generate "lvar = randomValue;"
for each abstraction predicate P_i:
  if P_i depends on lvar
    compute pre-images pre(P_i) and pre(¬P_i)
    compute under-approximation U(pre(P_i)) and U(pre(¬P_i)) in abstract domain
    generate "if U(pre(P_i)) b_i = true;
              else if U(pre(¬P_i)) b_i = false;
              else b_i = randomBool;"
```

Fig. 1. Abstracting an assignment statement: $lvar = rexp$ with respect to abstraction predicates P_i

The requirement can be lifted if we map the condition into abstract domain using the same computation as for converting pre-images into abstract domain. This will, however, introduce additional nondeterminism in the control flow of the abstract program. For instance, we could have an abstract if-statement with nondeterministic conditional branches like:

if (randomBool) then S_T else S_F;.

2.3 Dealing with object-orientedness

So far, we explored the way of computing abstraction for a state changing statement. This subsection explains how we extend the predicate abstraction in object-oriented paradigm.

For each class, the user can specify abstraction by defining predicates that depend on field variables in the class. We call this kind of abstraction *local* to a class. For verification of software programs, abstractions are often needed over small parts of the program. This local abstraction can be useful for abstracting subcomponents of a program when the whole program is too complicated to be abstracted globally.

Not only inside a single class but also over more than one class, the abstraction can be defined. There are many cases where abstract relations between variables from different classes are desired for the verification. We allow the user to specify new abstract variables which depend on variables

from two different classes. For example, if class A has a field x and class B has a field y then we can define an abstraction predicate $A.x > B.y$. We call this *inter-class* abstraction.

One of the major problems in dealing with object-oriented programs using the predicate abstraction is *referenced* variables. For local abstractions the abstraction variables are declared in the class which they belong to. Then, whenever the concrete variables in the definition of abstraction are modified in an object, we generate an abstract statement with the same reference to the object.

Unfortunately, building abstract program for inter-class abstraction is more involved than for the local abstraction in object-oriented paradigm. The abstract code needs to allow for many instantiations of objects of each different class to be handled correctly. For each inter-class abstraction, we generate an additional class in the abstract program, which contains a set of new boolean variables. The set of boolean variables correspond to the specified predicate for all the possible combinations of the instantiated objects in the relevant classes.

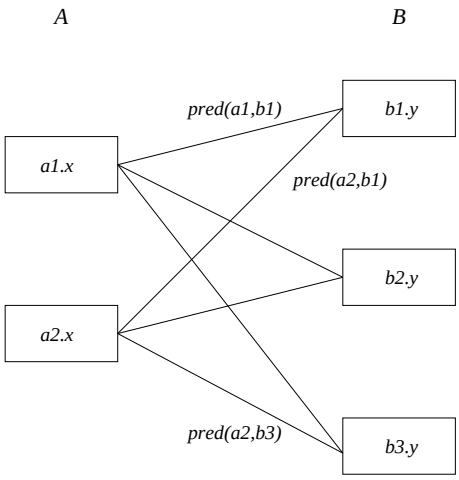

Fig. 2. Abstract Boolean variables for inter-class abstraction $A.x > B.y$

For instance, the set of predicates for an inter-class abstraction $B_I \equiv A.x > B.y$ is shown in Figure 2 and its new class definition in Figure 3. The two-dimensional array of "pred" contains the truth value of the predicate for each pair of objects of class A and of class B. The method setA() is called from the constructor of class A so that an instantiation of class

A will create new boolean variables with an index to the new object. The method getA() is used by the abstract statements to obtain the corresponding index of the "pred" for a given object or the corresponding object for a given index.

```
class BI {
   static public boolean[][] pred = new boolean[MAX][MAX];

   static public void setA(A objA){
      /* initialize abstraction variables and
         set index for a new object of class A */ }
   static public int getA(A objA){
      /* return the index for object objA */ }
   static public objA getA(int index){
      /* return the object objA at the index */ }
   static public void setB(B objB){
      /* initialize abstraction variables and
         set index for a new object of class B */ }
   static public int getB(B objB){
      /* return the index for object objB */ }
   static public objB getB(int index){
      /* return the object objB at the index */ }
}
```

Fig. 3. A class declaration for an inter-class abstraction

Whenever any variables in the definition of inter-class are modified, the corresponding set of the boolean variables need to be modified so that they are consistent with the values of concrete variables in the program. Suppose we have a statement in the concrete program $aobj.x++$, where $aobj$ is an object of class A. Because this assignment possibly changes the "pred" values of the inter-class abstraction B_I corresponding to the $aobj$, the an abstract assignment statement to the boolean variables $B_I.pred[\text{getA}(aobj)][*]$.

3 Automated Abstraction Tool

The abstraction tool is written in Java and uses the Java parser of Bandera system [5]. To use our abstraction tool, the user may specify the abstraction using methods of a special class "Abstract." To remove a field variable *var* in a class, the user annotates `Abstract.remove(var);` somewhere in the class. To add a new Boolean variable named *str* defined by a predicate *expr* in a class, the user annotates `Abstract.addBoolean("str", expr);` to the class. Inter-class abstraction can be declared using the same annotation where the variables in the definition are visible.

For computing abstraction, the tool uses Stanford Validity Checker (SVC) [1], which is an automated decision procedure to check the validity of formulas expressed in a subset of first-order logic. We transform logical expressions derived from Java statements (pre-images) into SVC format to use the validity checker. For example, for computing the abstract assignment (1) in the previous section, the logical expression of $x > 0 \Rightarrow x + 1 > 0$ is transformed into SVC format `(=> (> x 0) (> (+ x 1) 0))`, then passed to SVC through a foreign function call. The logic for SVC includes Boolean and uninterpreted functions and linear arithmetic and inequalities. Therefore, our abstraction tool cannot deal with any Java expression beyond this logic.

4 Example: Bakery Mutual Exclusion Algorithm

This section demonstrates how a real Java program is abstracted by our abstraction tool on the widely-used Bakery mutual exclusion algorithm. Figure 4 presents a part of the Bakery algorithm written in Java. The program has been annotated using `Abstract` methods for removing the unbounded integer `y1` in the `Process1` class and adding two Boolean variables, `y1EQ0` and `y1Pos` to the class. In addition, an inter-class abstraction variable `y1LTy2` has been defined in the main thread.

Figure 5, Figure 6, and Figure 7 in the following show the abstract Java program generated by the tool. First, Figure 5 shows the field variable declarations and their initial values for the new abstract variables `y1EQ0` and `y1Pos`, where SVC has been used to decide the initial values. The removed variable `y1` has been commented out and the unchanged variable `p2` remains the same. The constructor `Process1()` has been added in order to set the parameters of the inter-class abstraction variable `y1LTy2` whenever a Process1 object is instantiated. (See Section 2.3 for the details.)

Figure 6 shows the abstract code for the method `run()`. Each of three statements has been transformed into a set of statements encapsulated by `beginAtomic()` and `endAtomic()`. This atomicity is recognized by the JPF model checker. Note that, the abstract program can be longer than the original program but not necessarily has more states to be checked. The first set of statements replaces the original assignment y1 = p2.y2 + 1, which has been commented out because the variable `y1` no longer exists in the abstract program. The for-loop updates the inter-class abstraction variables in `y1LTy2` by the original assignment. Note that in the given Bakery program, there is only one such variable, `y1LTy2[0][0]`,

```
class Process1 extends Thread{
public int y1 = 0;
private Process2 p2;

void SetThread(Process2 p){ p2 = p; }

public void run(){
  Abstract.remove(y1);
  Abstract.addBoolean("y1EQ0", y1 == 0);
  Abstract.addBoolean("y1Pos", y1 > 0);

  while (true) {
    y1 = p2.y2 + 1;
    while (p2.y2 != 0 && y1 >= p2.y2);
    y1 = 0;
} } }
```

```
class Process2 extends Thread{
public int y2 = 0;
private Process1 p1;

void SetThread(Process1 p){ p1 = p; }

public void run(){
  Abstract.remove(y2);
  Abstract.addBoolean("y2EQ0", y2 == 0);
  Abstract.addBoolean("y2Pos", y2 > 0);

  while (true) {
    y2 = p1.y1 + 1;
    while (p1.y1 != 0 && y2 > p1.y1);
    y2 = 0;
} } }
```

```
class Bakery {
  public static void main(String args[]){
    Process1 process1 = new Process1();
    Process2 process2 = new Process2();
    process1.SetThread(process2);
    process2.SetThread(process1);

    Abstract.addBoolean2("y1LTy2", process1.y1 < process2.y2 );

    process1.start();
    process2.start();
} }
```

Fig. 4. Bakery mutual exclusion algorithm in Java

```
class Process1 extends Thread{
  // public int y1 = 0;
  public boolean y1EQ0 = true;  // SVC valid: (=> (and (= y1 0)) (= y1 0))
  public boolean y1Pos = false; // SVC valid: (=> (and (= y1 0)) (not (> y1 0)))
  private Process2 p2;
  void SetThread(Process2 p){ p2 = p; }

  // see the next figure for run().

  Process1() { y1LTy2.setProcess1(this); }
}
```

Fig. 5. Abstracted Java program of the Bakery mutual exclusion algorithm (1)

```
public void run(){
  while (true){
    Verify.beginAtomic();
      // y1 = p2.y2 + 1;
      for(int i = 0; i < y1LTy2.numProcess2; ++i){
        if( i == y1LTy2.getProcess2(p2) )
          y1LTy2.pred[y1LTy2.getProcess1(this)][i] = false;
        else y1LTy2.pred[y1LTy2.getProcess1(this)][i] = Verify.randomBool();
      }
      if(p2.y2EQ0 || p2.y2Pos) y1EQ0 = false;
      else y1EQ0 = Verify.randomBool();
      if(p2.y2EQ0 || p2.y2Pos) y1Pos = true;
      else y1Pos = Verify.randomBool();
    Verify.endAtomic();

    while (!p2.y2EQ0 &&
           !y1LTy2.pred[y1LTy2.getProcess1(this)][y1LTy2.getProcess2(p2)]){};

    Verify.beginAtomic();
    // y1 = 0;
    for(int i = 0; i < y1LTy2.numProcess2; ++i){
      if(y1LTy2.getProcess2(i).y2Pos)
        y1LTy2.pred[y1LTy2.getProcess1(this)][i] = true;
      else if(y1LTy2.getProcess2(i).y2EQ0 || !y1LTy2.getProcess2(i).y2Pos)
        y1LTy2.pred[y1LTy2.getProcess1(this)][i] = false;
      else y1LTy2.pred[y1LTy2.getProcess1(this)][i] = Verify.randomBool();
    }
    y1EQ0 = true;
    y1Pos = false;
    Verify.endAtomic();
} }
```

Fig. 6. Abstracted Java program of the Bakery mutual exclusion algorithm (2)

because only one instance has been created for each class of Process1 and Process2. The following two if-statements updates the local abstraction variables. The condition of the while-statement has been simply replaced by the matching abstraction predicate[2]. Finally, the last set of statements corresponds to the original assignment y1 = 0.

```
class y1LTy2 {
  static final int MAX = 3;
  static public boolean[][] pred = new boolean[MAX][MAX];

  static public int numProcess1 = 0;
  static public Process1[] objProcess1 = new Process1[MAX];
  static public void setProcess1(Process1 obj){
    objProcess1[numProcess1++] = obj; }
  static public int getProcess1(Process1 obj){
    for(int i = 0; i < numProcess1; ++i)
      if(obj == objProcess1[i]) return i;
    return MAX + 1; }
  static public Process1 getProcess1(int i){
    return objProcess1[i]; }

  static public int numProcess2 = 0;
  static public Process2[] objProcess2 = new Process2[MAX];
  static public void setProcess2(Process2 obj){
    objProcess2[numProcess2++] = obj; }
  static public int getProcess2(Process2 obj){
    for(int i = 0; i < numProcess2; ++i)
      if(obj == objProcess2[i]) return i;
    return MAX + 1; }
  static public Process2 getProcess2(int i){
    return objProcess2[i]; }
}
```

Fig. 7. Abstracted Java program of the Bakery mutual exclusion algorithm (3)

Figure 7 shows a class definition for the inter-class abstraction predicate `y1LTy2`. The truth values of the predicates are maintained in the array variable `pred[][]` as explained in Section 2.3. The object references `Process1[]` and `Process2[]` and the initial value of the predicates are set by the corresponding constructors using the `setProcess1()` and `setProcess2()` methods. The `getProcess1()` and `getProcess2()` methods are called from the abstract assignment statements to refer the corresponding `pred[][]` abstract variables.

[2] We currently require the user to provide the same predicate as an abstraction criterion as explained in Section 2.2

5 Conclusion

We applied and extended predicate abstraction for object-oriented programs. Using the techniques, we have been able to obtain abstract Java programs of several examples automatically. Moreover, use of the abstraction tool allows for a sound approximation of the concrete program using an automated validity checker and it helps to avoid error-prone abstraction by human reasoning.

However, the technique does not necessarily renders the most accurate abstract interpretation of a given program and the user must give a reasonable abstraction guidance to generate a meaningful abstract program for checking desired properties. If the guidance is not good enough, the result will be a too coarse abstract program which can not preserve the properties being checked.

Future work includes extension of the inter-class abstraction for more than two classes and constructing multi-valued abstract variables from a set of predicates. We are currently working on solutions for additional aliasing problem due to arrays and abstraction involving array variables.

Acknowledgment

We thank Clark Barrett, Satyaki Das, and David Dill at Stanford University for their help on SVC, and John Hatcliff, Corina Pasareanu, and Robby at Kansas State University for providing the Java parser in Bandera system.

References

1. C. Barrett, D. Dill, and J. Levitt. Validity checking for combinations of theories with equality. In *Formal Methods In Computer-Aided Design*, volume 1166 of *Lecture Notes in Computer Science*, pages 187–201. Springer-Verlag, November 1996.
2. M. Colón and T. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In *Proceedings of the 10th Conference on Computer-Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 293–304. Springer-Verlag, July 1998.
3. S. Das, D. Dill, and S. Park. Experience with predicate abstraction. In *Computer Aided Verification, 11th International Conference, CAV'99, LNCS 1633*, pages 160–171, July 1999.
4. S. Graf and H. Saïdi. Construction of abstract state graphs with pvs. In Orna Grumberg, editor, *Computer Aided Verification, 9th International Conference, CAV'97*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83, Haifa, Israel, June 1997. Springer-Verlag.

5. John Hatcliff, Matthew Dwyer, and Shawn Laubach. Staging static analyses using abstraction-based program specialization. In *Proceedings of Principles of Declarative Programming: 10th International Symposium, PLILP'98*, September 1998. LNCS 1490.
6. Kedar Namjoshi and Robert Kurshan. Syntactic program transformations for automatic abstraction. In *Computer Aided Verification, 12th International Conference, CAV'2000, LNCS 1855*, pages 435–449, July 2000.
7. H. Saïdi and N. Shankar. Abstract and model check while you prove. In *Proceedings of the 11th Conference on Computer-Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 443–454. Springer-Verlag, July 1999.